Making a System call in xK (see usys.S + syscall.c)


```
main:
    :
    movq $15, %rax
    int  $64
    :
```

- calls interrupt handler with trapno = 64
  - ↳ calls syscall() (with %rax = 15 in trapframe)
    - ↳ calls sys_open

System calls @ C level

"pair of stubs"

User space

```
main() {
    :
    int fd = open(arg1, arg2);
    :
}
```

```
open() {    //stub
    movq $15, %rax
    int $64
    retq
}
```

prev slide

Kernel space

```
actually_open(...) {
    // do the operation...
}
```

```
sys_open() {  // stub
    // checks...
    // copy args into kernel
    actually_open(...);
    // copy ret val out of kernel
    return;
}
```

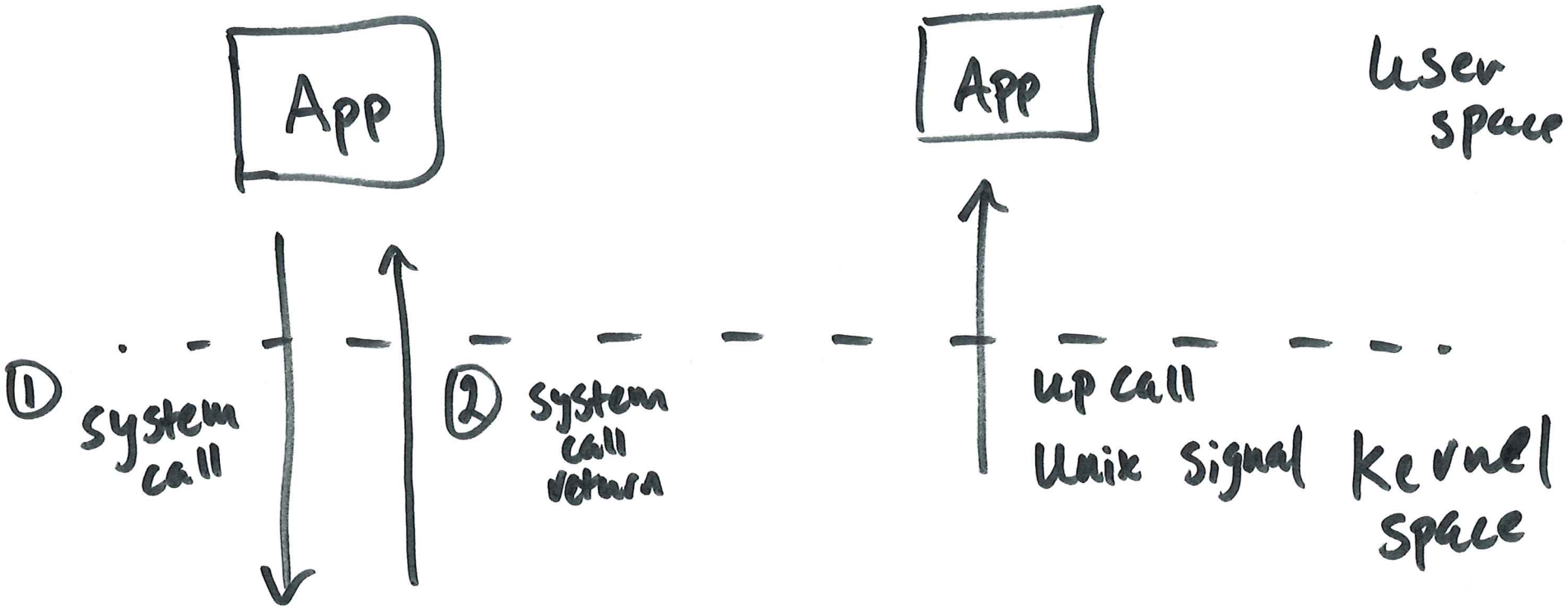
Securing System Calls

- danger: app asks Kernel to do something
  - Kernel has permission to do anything
  - must manually check that app has perms.
- example: app passes virtual address that points to Kernel memory
  - Kernel has permission to access that addr!
  - app does not.
- example: copy before check perms.
  - otherwise memory might change (!)

3.16

Upcalls + Virtual Machines

Upcalls: like backwards system calls

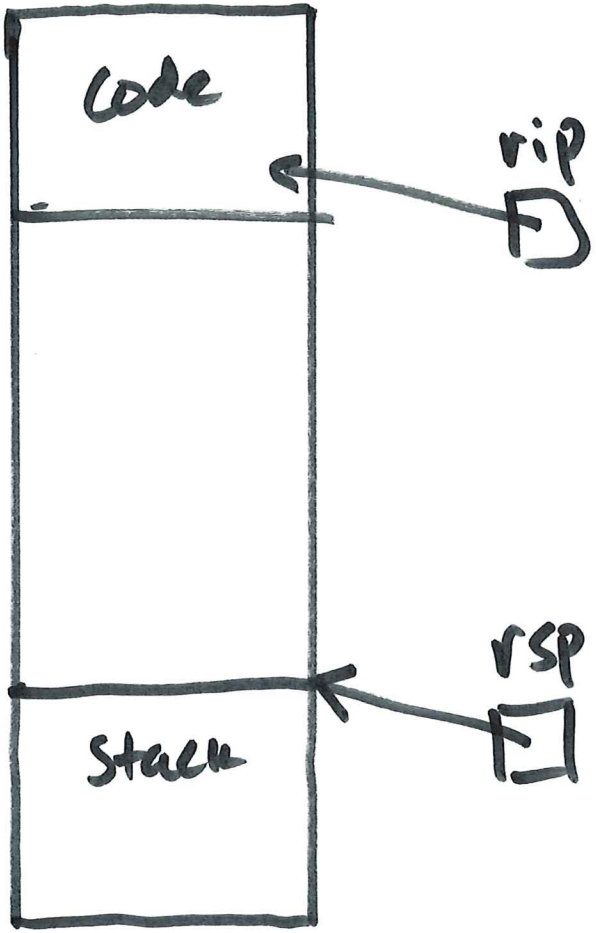


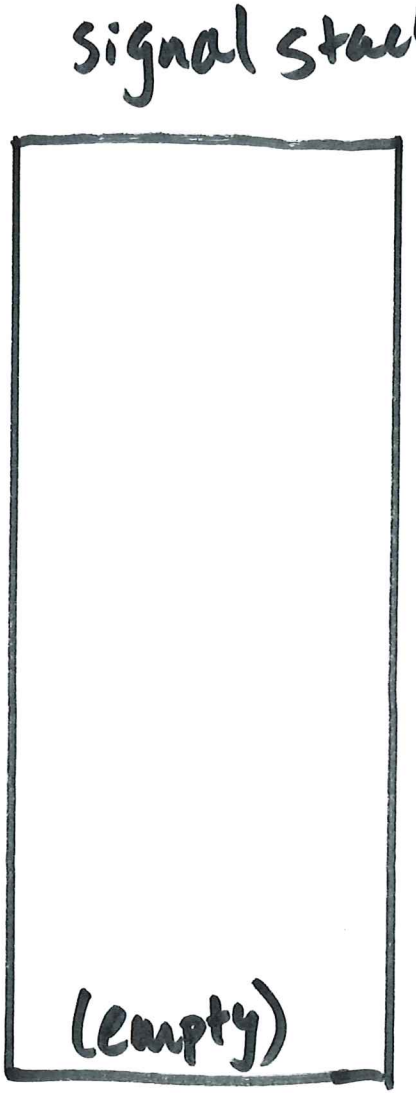
Why upcall?

- asynchronous I/O notification
  - syscall to enqueue op, signal to notify complete
- user-level exception handling
  - e.g. save files before exit
- user-level thread libraries: timer events

Generally: upcalls also allow user space to become more like kernel "virtual interrupts"

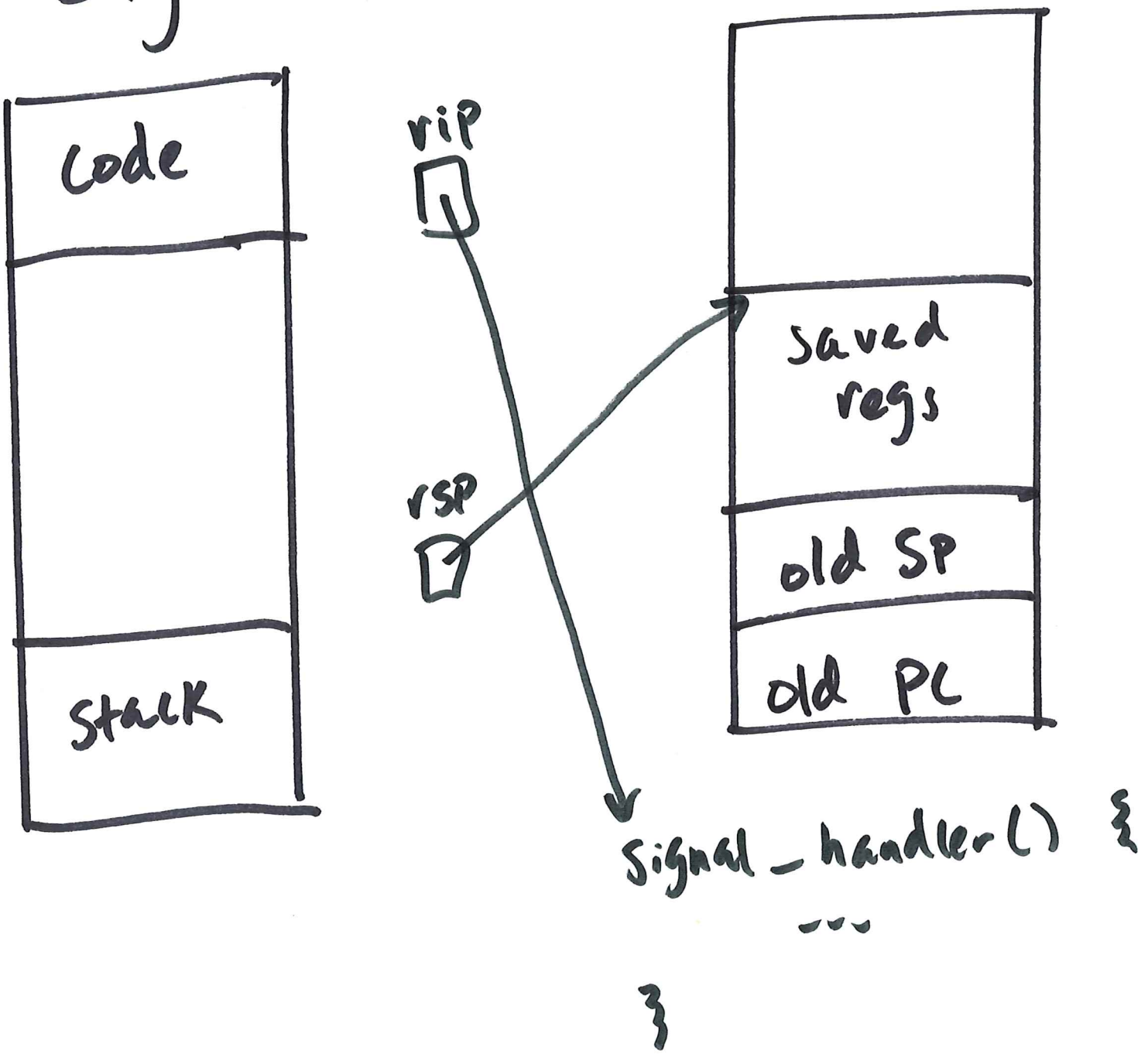
Unix Signals






```
signal_handler () {
...
}
```

Unix Signals



Signals as virtualized interrupts

interrupts

interrupt number

interrupt handler

interrupt stack

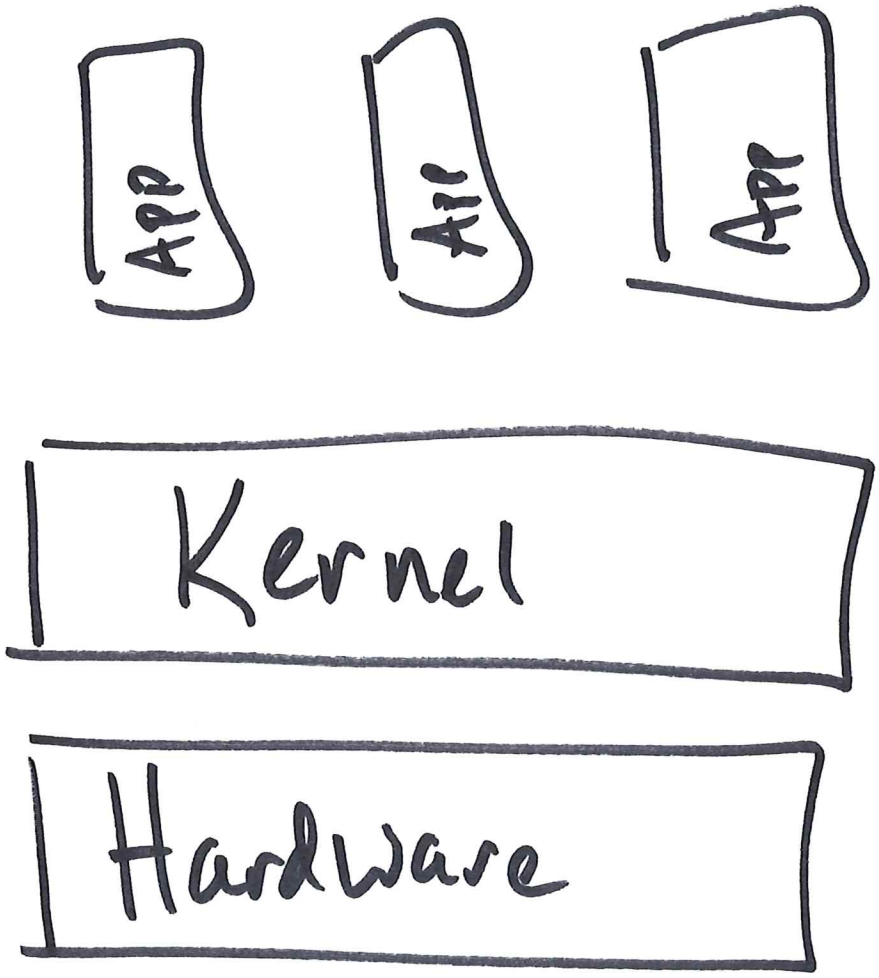
Signals

signal number

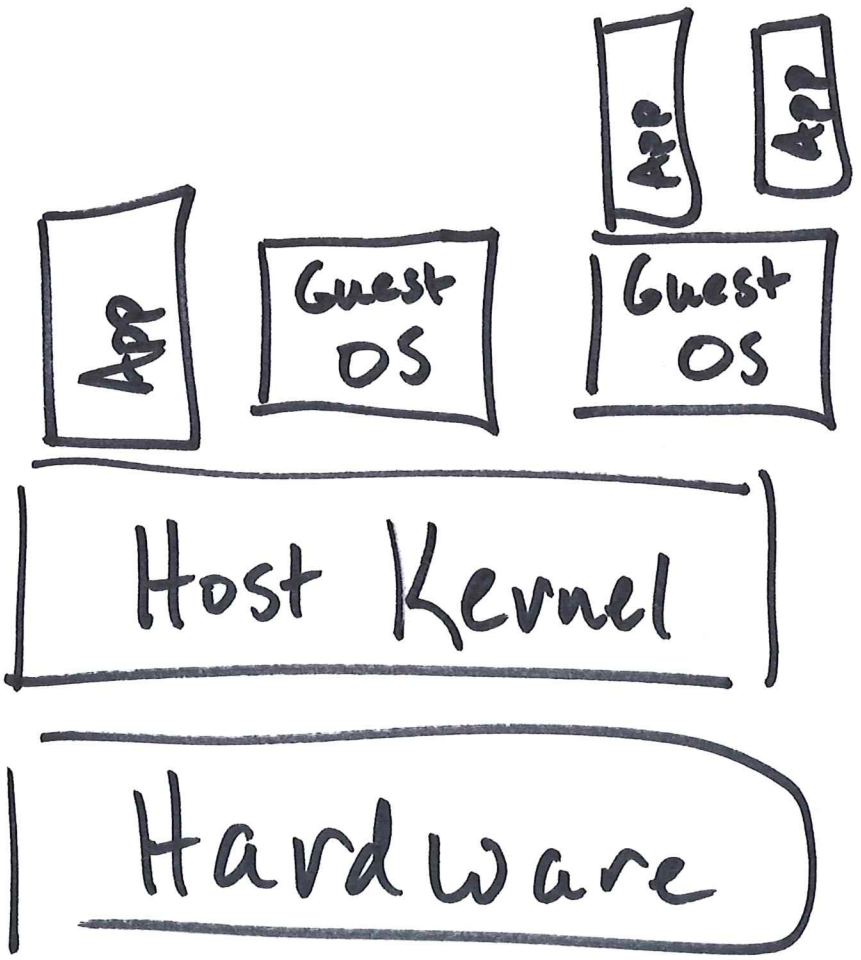
signal handler

signal stack

OS world



Virtual Machine World

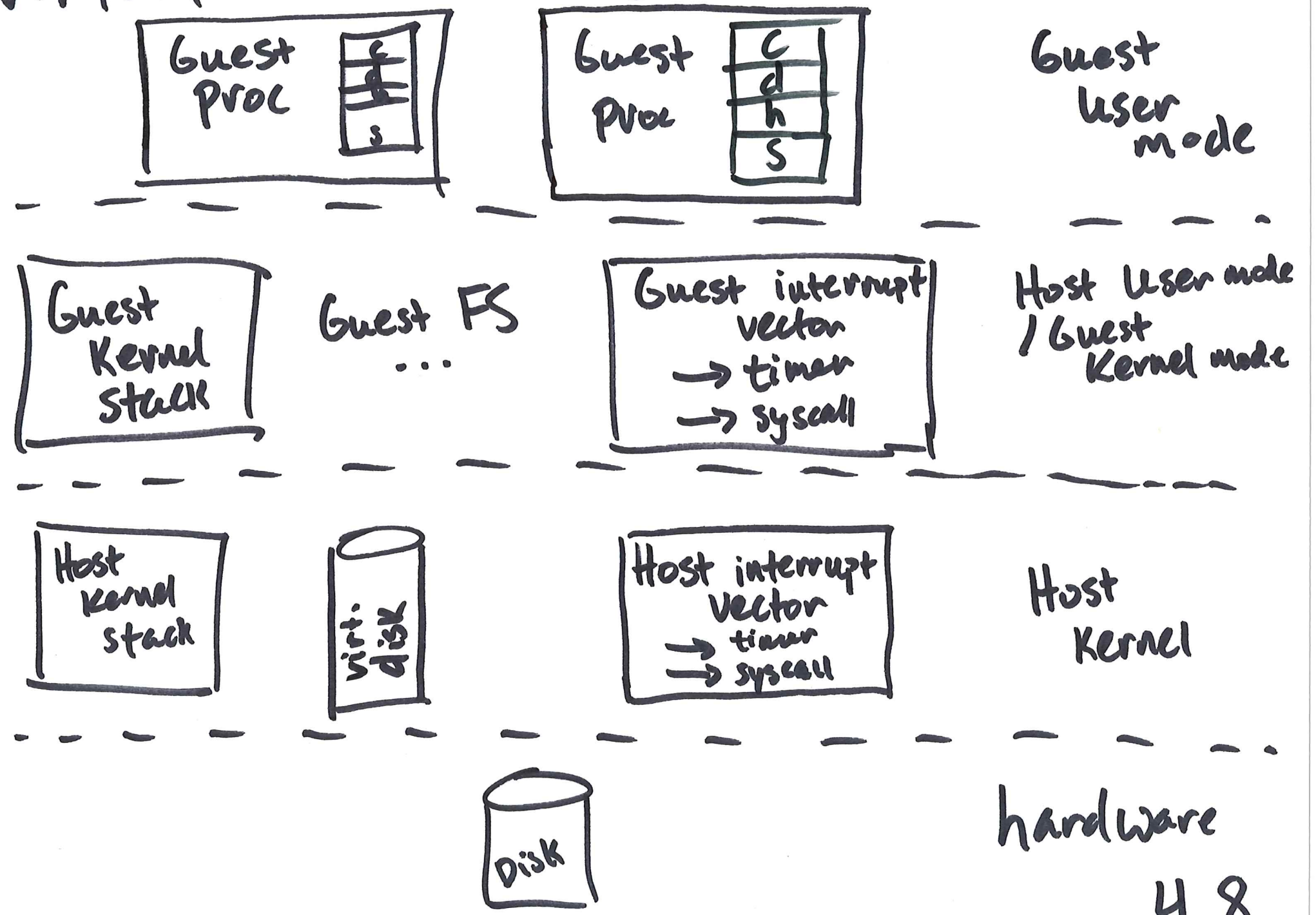


Why VM?

- isolate application environments
  - e.g. legacy apps
- efficient use of server hardware
  - run multiple VMs on one physical mach.
- transparent hardware upgrade
  - save entire state of VM and resume
- OS debugging

4.7 1/2

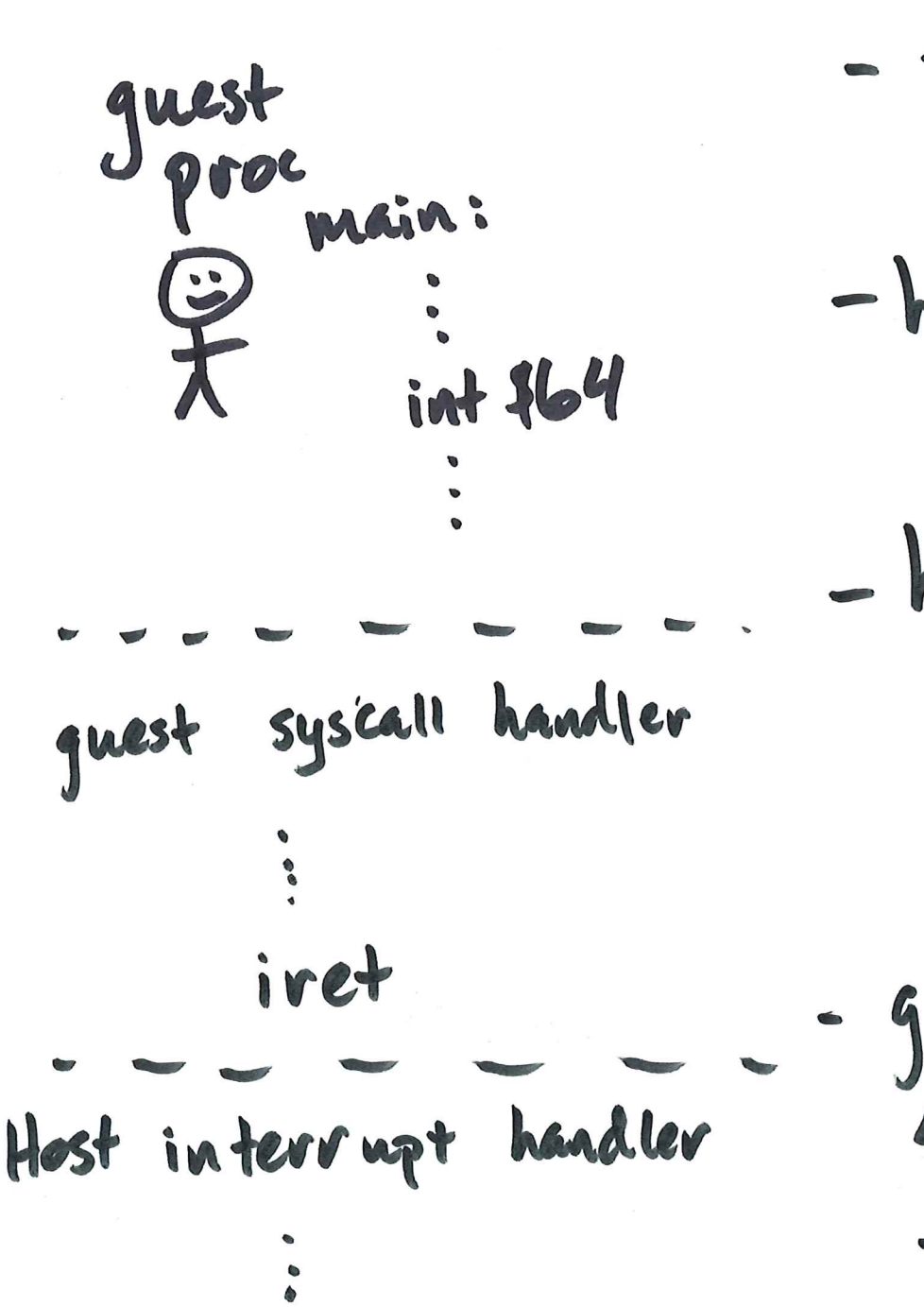
Virtual Machines (see slide 1.3)



Trap + Emulate

- Guest OS runs in user mode!
  - What happens when it executes privileged instruction?
- In theory, all such instructions trap into Host OS
  - Host OS can then emulate the instr. to guest OS

Guest App system call



- traps to host OS interrupt handler (!)
- host recognizes source was inside VM
- host emulates hardware behavior for Guest OS
  - save pc/sp, call guest interrupt handler
- guest OS executes iret
  - ↳ priv. instr. traps to host OS
- host emulates hardware
  - restore pc/sp, resume app